EXHIBIT 2

Implicit Networks, Inc.
U.S. Patent No. 6.629,163 C1
Claims Chart
Implicit Networks, Inc. v. Juniper Networks, Inc.
Flow Based Processing

'163 C1 Patent: Claim 1	Method and System for Demultiplexing a First Sequence of a Packet, Components to Identify Specific Components, Wherein Subsequent Components Are Processed Without Re-Identifying Components

OVERVIEW

This overview consists of three sections, and is written (save the source code excerpts) in plain English, to set forth, clearly and directly, Implicit's infringement contentions.

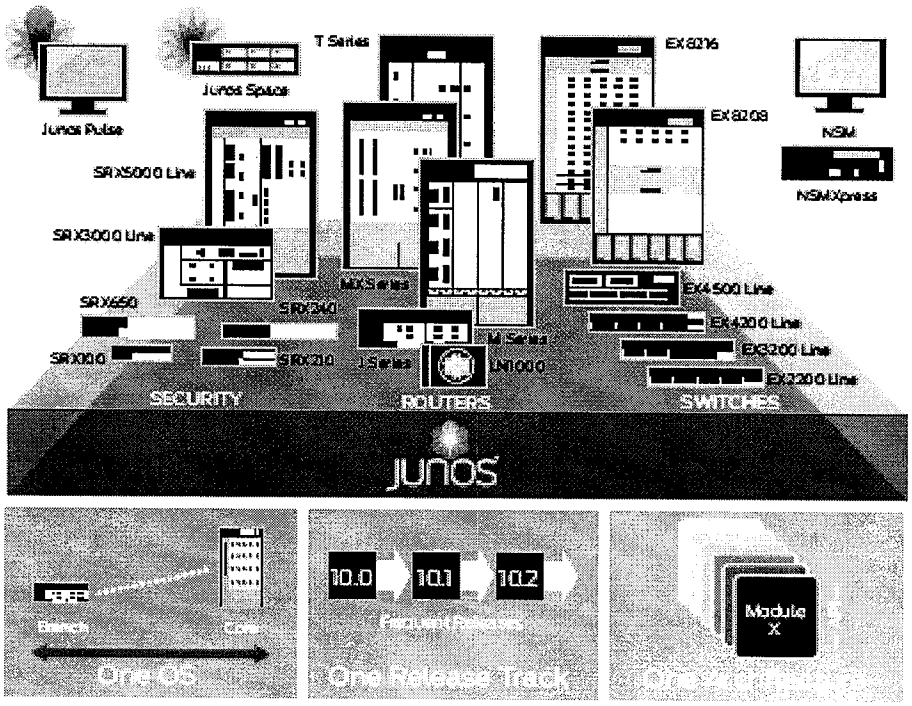
Section 1 provides a general overview. Section 2 describes, step by step, how the Juniper Networks, Inc. ("JNI") system actually processes the first packet of a new flow, and thereafter subsequent packets of the same flow. Section 3, in summary form, maps Implicit's claims to JNI's architecture, along with additional pinpoint citations. Thereafter, the charts provide additional detail and evidentiary support for this narrative summary.

There are three exhibits attached: (1) a full list of the JNI individual code modules, known as plug-ins; (2) an index mapping code functions to bates stamp production pages; and (3) a list mapping the full file and path names in the code to bates stamped pages produced.

1. JNI's Integrated Operating System: JUNOS.

Juniper's operating system, known as JUNOS, is an integrated operating system, common to many but not all Juniper products. *See* Disclosure, Appendix A, "Accused Products." As Juniper describes it, the JUNOS operating system provides "a common language across Juniper's routing, switching, and security devices…." As an important competitive point of differentiation, Juniper designed JUNOS to be "one operating systems delivering one software release track with one modular architecture." *See* JUNOS OS: The Power of One Operating System, at 1.

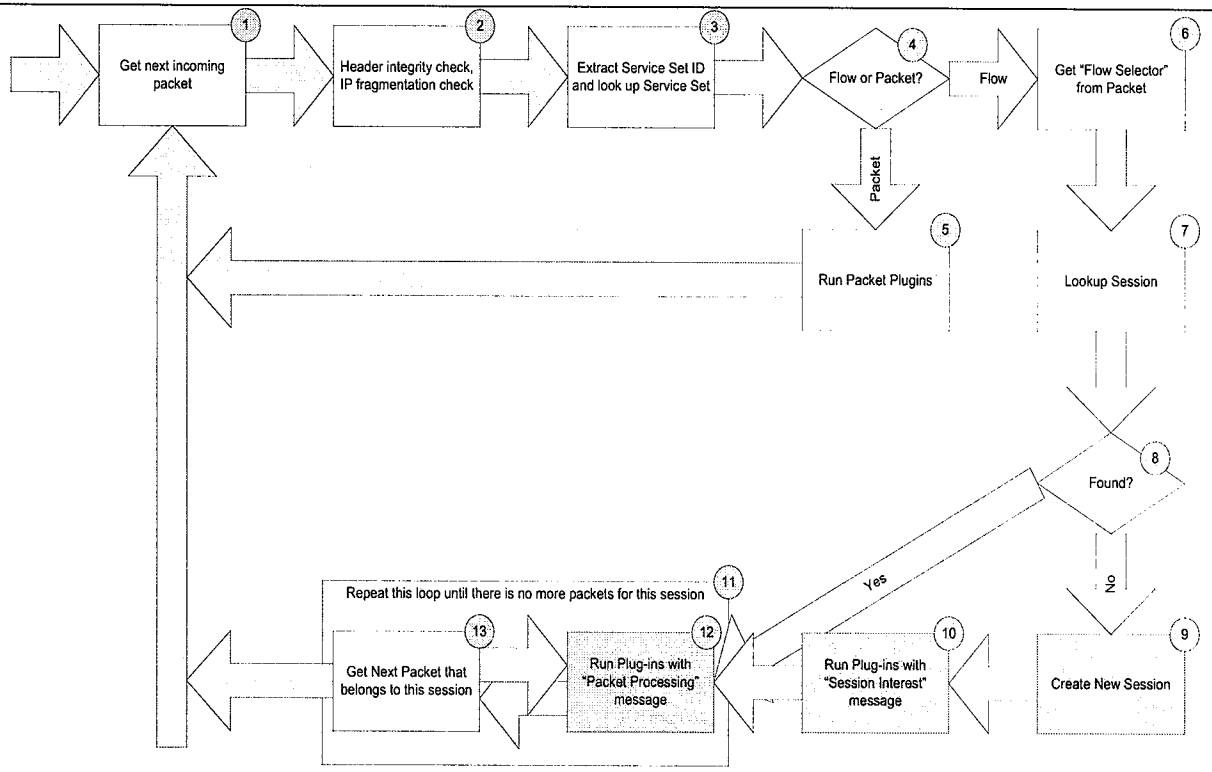
As Juniper captures this graphically:



As this graphic illustrates, many JNI products run on the common operating system, JUNOS.

JUNOS offers flow-based (stateful) packet processing. In contrast to packet based forwarding, where every packet stands alone without regard to flow or state information, flow-based processing requires the creation of sessions. A session is created to store, *e.g.*, the security measures to be applied to the packets of the flow, to cache information about the state of the flow, to allocate required resources for the flow, and to

	provide a framework for features such as Application Layer Gateways (ALG's) and firewall features. *See* Understanding Packet-Based and Flow-Based Forwarding, at 1-2. In the flow-based functionality, JUNOS classifies the first packets of a new flow through at least a five tuple classification process. If the packet is identified as the first packet of a new flow, it goes through a policy lookup, whereby the system applies configuration information to the new packet to determine how that flow should be processed. The system then creates an instantiated and stateful data processing path, employing code modules called plugins. The first and subsequent packets of the flow are run through this dynamically created data processing path, and state is maintained accordingly. This approach gives the accused JNI products the ability to provide stateful firewall functionality, ALG function, NAT, and other applications of flow based processing. *See* Exhibit 1 (List of Plug-Ins) and JUNOS, Feature Support Reference for SRX Series and J Series Devices, which describe these functions. Put graphically, JNI's basic packet processing loop is as follows:



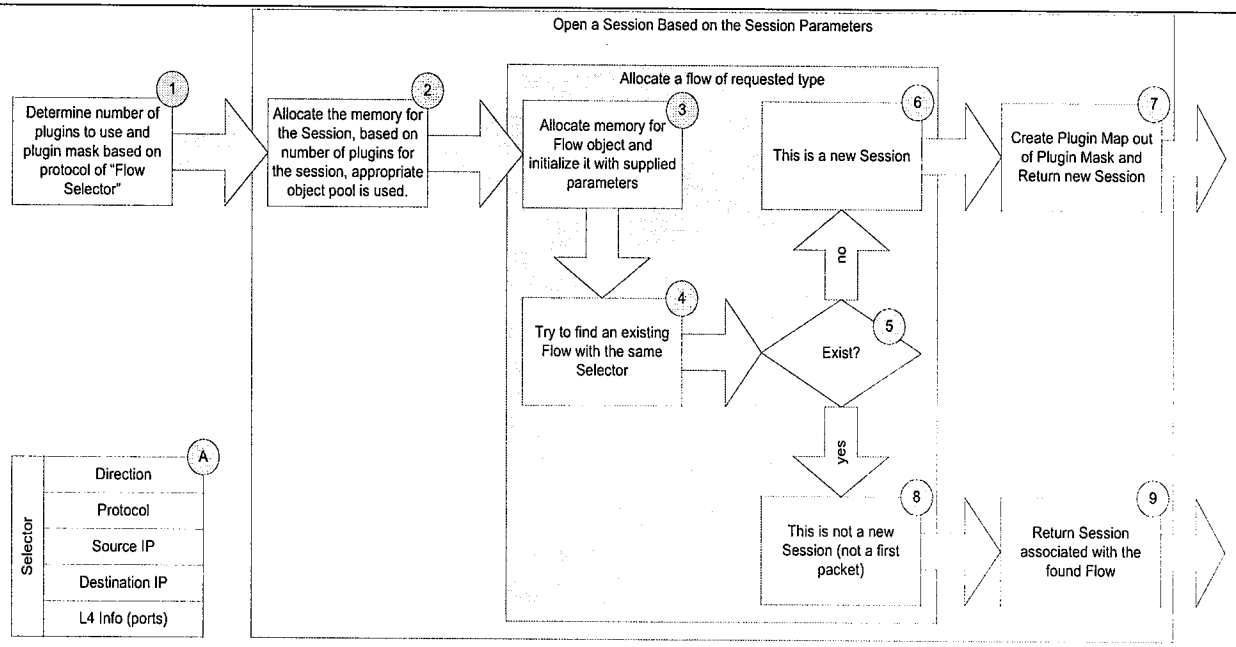
This graphic foots to specific code citations as follows:

1. The next packet is extracted from the queue of incoming packets (Juniper Source Code ("JSC") 00001061, line: 1046);
2. Various checks are performed, such as Header integrity check (JSC00001061, line: 1054) and IP fragmentation check (JSC00001061, line: 1062).

	3. Service Set ID is extracted from the packet meta-data (JSC00001062, line: 1080) and Service Set object is looked up for the extracted Service Set (JSC00001063, line: 1125). 4. Determine the Flow Type of the Service Set ("flow" or "packet") (JSC000001064, line: 1156). 5. In case it is a "packet" flow type – run packet plugins (JSC00001064, line: 1164) and jump to Step #1 above. 6. In case it is a "flow" flow type – extract Flow Selector (JSC00001064, line: 1175) from the packet (where Flow Selector is a structure containing Direction of the Flow, Protocol, Source and Destination IP addresses and Source and Destination Ports). 7. Try to locate an existing session (JSC00001065, line: 1184) based on Flow Selector. 8. If an existing session has been found – jump to Step 11 below. 9. Create new Session object (JSC00001065, line: 1208). (*See* detailed description of this process in the Section "New Session Creation" below). 10. Run the chain of plugins (JSC00001068, line 1319) assigned to this Service Set with "Session Interest" message and the packet being processed. (*See* detailed description of this process in the Section "Run Plugins" below). 11. Repeat Steps 12 and 13 below for every subsequent packet associated with this session. 12. Run the chain of plugins (JSC00001070, line: 1383) assigned to this Service Set with "Packet Processing" message and the packet being processed. (*See* detailed description of this process in the Section "Run Plugins" below). 13. Get the next packet associated with this session (JSC00001071, line: 1423). In JUNOS flow-based configurations, the treatment of the packets in the flow depends on the characteristics for the first packet in that flow. That is, the **what** of it, the content, drives the **how** of it, the processing. **2. First Packet Methodological Walk Through.** When a packet arrives at the system, it is classified according to multiple classification criteria, *e.g.*, source address, destination address, source port, destination port, and other such classification

	characteristics. This can often be a greater than a five tuple classification process. If this process identifies the packet as the first packet of a new flow, as against a packet belonging to a flow already transversing the system (*see* below), the system undertakes a policy look-up to determine what actions need to be taken, *i.e.*, what processing steps should be taken as to that particular flow. Each separate service, *e.g.* firewall, or antivirus, has a separate policy look-up. The processing steps called for by the policies are captured in independent C-file modules. The system comes with modules (actions) loaded, and the policies call out which modules (actions) will be taken. The system administrator will load configuration information, *e.g.*, what the admin wants the system to do with particular types of traffic. For instance, the admin could, for a particular type of traffic, specify that the system should have a firewall feature, compression, and the like. The admin makes a human readable list of the desired actions, which the JUNOS system turns into a binary "plug-in mask." A plug-in mask in an internal representation of a list of the plug-ins to be used for the specified traffic, *i.e.*, configuration information. When the traffic comes in, there is a policy look-up, the plug-in mask (list) found, and a stateful data processing path then instantiated with memory allocated, as per the configuration information and as based on packet inspection. In this fashion, the system determines which actions (processing steps) should be taken and which not. After the policy look-ups are undertaken for the first packet, the actions to be taken are stored in the data structure called a flow state. The status of one particular flow is called a flow state; the status of all of the flows transversing the system is held in a data structure known as a flow table. When subsequent packets of the same flow arrive, they are classified, associated with the existing flow state, and the actions performed on the first packet are subsequently performed on the subsequent packets. The actions to be taken are stored as data structures in memory, post-first packet inspection. In this way, the policy look-up need not occur on a recursive packet-by-packet basis for a given flow. In the JUNOS architecture, policies can be changed dynamically at run-time without shutting down the system, recompiling the kernel, and spooling the system up. Indeed, the system is designed to be modular, extensible, and changeable at run-time.

	New Session: Creating a Data Processing Path Based on Information in the First Packet Central to the processing process is what JNI calls new "session creation" for the first packet of a new flow. By step, New Session Creation consists of the following, with pinpoint code citations: 1. Determine (JSC00001044, line: 399) number of plugins to use and plugin map based on protocol of "Flow Selector" (A). 2. Allocate the memory for the Session (JSC00001138, line: 481), based on number of plugins for the session – appropriate object pool is used. 3. Allocate memory for Flow object and initialize it with supplied parameters (JSC00001121, line: 827) 4. Try to find an existing Flow with the same Selector (JSC00001122, line: 850) 5. If an existing Flow was found (JSC00001139, line: 512) – go to Step 8 below; otherwise go to Step 6. 6. This is a new Session and the packet that triggered the session creation is the first packet. 7. Create Plugin Map out of Plugin Map and Return new Session. 8. This is an existing Session and the packet is not the first one. 9. Return Session associated with the existing Flow (JSC00001139, line: 517). The Diagram below and the description that follows outline the steps taken during New Session Creation. This is an expansion of Step 9 in "Main Packet Loop" discussion above.



Running the Code Modules (Plugins)

Another central aspect of the JNI system lies in what it calls "running the plug-ins," *i.e.*, running data through the dynamically created data processing path. A "plug-in" is a code module that does a particular thing. Typical tasks performed by plug-ins include the following:

- Creating object caches for memory allocation using the msp_objcache_create function.
- Attaching state to sessions using extension handles.
- Attaching timers to sessions.
- Influencing packet flow using packet actions and session actions in response to system-

generated events.

- Using predefined session types, allowing multiple plug-ins to share the same session.
- Creating new sessions if needed, using new flow types.

A list of plug-ins is attached as Exhibit 1.

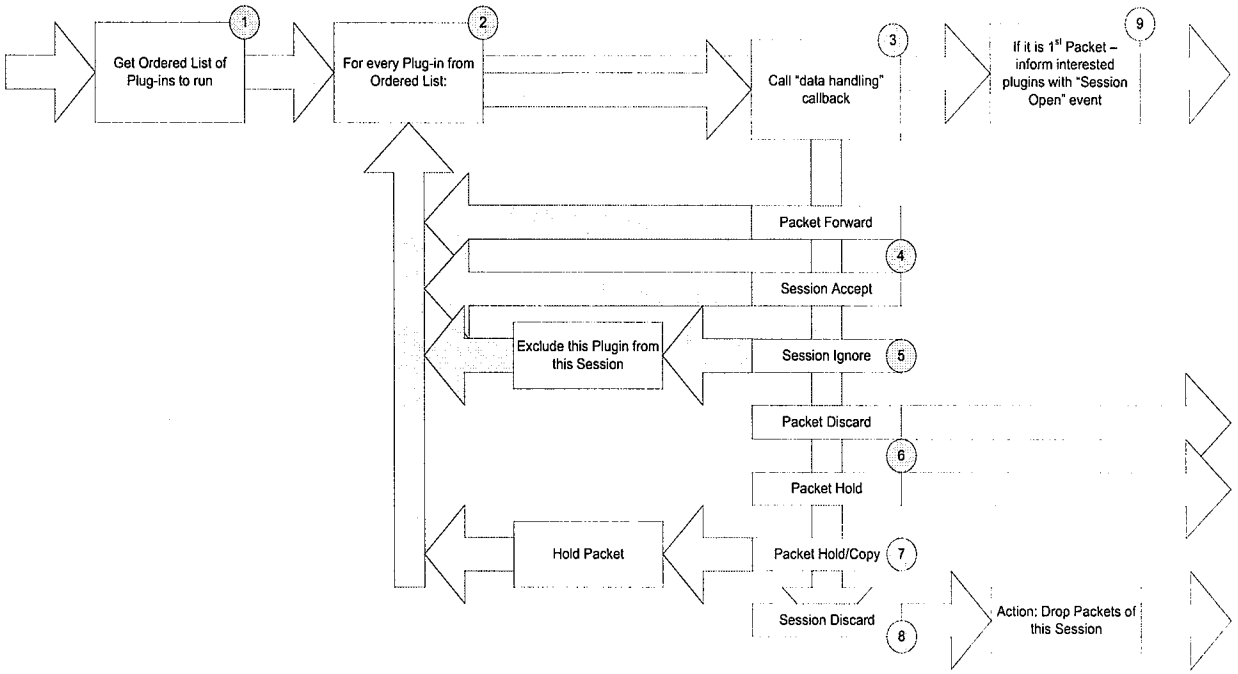
Plug-ins are designed to be independent and individual. Each performs a rule match on its own packet processing function, independent of other plug-ins, and saves state independently.

By step, the running of the plug-in process proceeds as follows:

1. An ordered list of the Plugins to be executed is obtained (JSC00001049, line: 596) from the Service Set; *see* plug-in discovery and configuration below;
2. For every plugin from the list obtained, performed the following operations, unless the plugin has indicated that it should be excluded from packet processing within this session (JSC00001051, line: 650);
3. Call Plugin's "data handling" function (JSC00001053, line: 726), and analyze the result:
4. If the result of the previous step was "packet forward" or "session accept" – move to the next plugin in the chain of plugins (go to Step 2 above);
5. If the result of the Step 3 was "session ignore" – exclude this plugin from future processing of the packets of the current session (JSC00001054, line: 763) and move to the next plugin in the chain of plugins (go to Step 2 above);
6. If the result of the Step 3 was "packet discard" or "packet hold" – stop calling any other remaining plugins and return the result;
7. If the result of the Step 3 was "packet hold/copy" – change the value of the result that the function will return to "hold packet" (JSC00001054, line: 788) and move to the next plugin in the chain of plugins (go to Step 2 above);
8. If the result of the Step 3 was "session discard" – inform the flow to drop all subsequent packets associated with this flow (JSC00001054, line: 775), and return the result.

9. After exhausting the list of plugins, and in case we were processing the first packet of the session – notify all the plugins with "session open" event" (JSC00001054, line: 805).

The Diagram below captures this process graphically; this is an expansion of Steps 10 and 12 in the Section, ("Main Packet Loop").



How Plug-Ins Are Discovered and Loaded

Loading

Plug-ins are discovered by mspman daemon during its startup process in the following fashion: A directory (`/opt/sdk/pkg/`) is searched for plug-ins configuration files (*.xml). An example of a configuration file is provided below:

```
<pkg-config>
    <!-- Package configuration header - one per config file -->
    <header>
       <!-- config file syntax version -->
       <version>1.0</version>
       <!-- unique services package (container) name -->
       <name>sample-services</name>
    </header>

    <!-- List of plugins -->
    <plug-in>
       <!-- unique plugin package name -->
       <name>sample-foo</name>
       <!-- plugin shared object full name (with path) -->
       <path>/opt/sdk/lib/sample-foo.so.1</path>
       <!-- plugin entry point, this function is called when plugin src is loaded -->
       <entry>sample_foo_entry</entry>
    </plug-in>

    <plug-in>
       <!-- unique plugin package name -->
       <name>sample-bar</name>
       <!-- plugin shared object full name (with path) -->
       <path>/opt/sdk/lib/sample-bar.so.1</path>
       <!-- plugin entry point, this function is called when plugin src is loaded -->
       <entry>sample_bar_entry</entry>
    </plug-in>
</pkg-config>
```

In the example above, two plug-ins (`sample-foo` and `sample-bar`) are described by specifying their location and the name of the entry function to be called to initialize the plug-in.

Mspman daemon dynamically loads specified modules (e.g. `sample-foo.so.1, sample-bar.so.1`) and calls the entry function specified by the configuration file (e.g. `sample_foo_entry, sample_bar_entry`).

Within the entry point function (implemented by plug-in, and called by mspman daemon), plug-in registers itself by calling `msvcs_plugin_register` function, and providing an instance of `msvcs_plugin_params_t` structure (see below) filled with plug-in specific information, which includes (besides other information) the pointers to the functions to be called in response to incoming packets and control events (event handlers).

In other words, each plug-in exists as a dynamic object, containing three components: entry point, control event handler and data event handler.

- The entry point is called by the management daemon when it loads the plug-in as a dynamic object.
- The control event handler is called by the plug-in when the service receives an initialization or configuration data-received event.
- The data event handler is called by the plug-in when the service receives an event related to packets.

3. Claims Mapping: JUNOS Infringement.

In addition to the source citations and narrative above, following is a claims mapping:

Claim	Citation
A method in a computer system for processing a message having a sequence of packets, the method comprising:	JunOS Operating System
providing a plurality of components, each component being a software routine for	*See* Attachment 1 for the list of plug-ins.

	converting data with an input format into data with an output format;	Example of data conversion: • Compression/decompression, • Encryption/decryption.
	for the first packet of the message,	*See* example below, where function `cpcd_process_session_interest` (JSC00001004, line 185) is called in response to receipt of the first packet.
	dynamically identifying a non-predefined sequence of components for processing the packets of the message such that the output format of the components of the non-predefined sequence match the input format of the next component in the non-predefined sequence, wherein dynamically identifying includes selecting individual components to create the non-predefined sequence of components after the first packet is received; and	A data processing path is created from the use of configuration information (plug-in list; the recipe) post-first packet and based on the information in the first packet; *see* code section in overview above; Example of a plugin indicating the need for exclusion from future processing: function `cpcd_process_session_interest` (JSC00001004, line 185) in multiple cases return `MSVCS_ST_SESSION_IGNORE` value. Remembering the fact of exclusion: function `mspman_svcs_run_plugins` (JSC00001048; line 543) upon receipt of the `MSVCS_ST_SESSION_IGNORE` return value from calling the chain of plugins – invokes function `msvcs_session_ignore` (JSC00001054; line 763), which in turn saves plugin's ID and direction of the flow (JSC00001156; line 1175) for which the exclusion should be enacted. Excluding in action: function

		`mspman_svcs_run_plugins` (JSC00001048; line 543) before invoing an appropriate plugin's event handling function checks if this particular plugin was excluded from future involvement (JSC00001051, line 649) and if it was – the plugin is not called (JSC00001051, line 650).
	storing an indication of each of the identified components so that the non-predefined sequence does not need to be re-identified for subsequent packets of the message; and	"Session" object has special field "`sn_ignore`" (JSC00001357, line 157), that holds information (plugin ID and flow direction) about plugins that should be excluded from originally defined sequence.
	for each of a plurality of packets of the message in sequence,	While processing each packet, …
	for each of a plurality of components in the identified non-predefined sequence,	… each plugin's event handling function …
	retrieving state information relating to performing the processing of the component with the previous packet of the message;	… has access to the session object, which in turn has a dedicated array of opaque (plugin dependent) storage elements - `sn_handles` (JSC00001357, line 163), where plugin can create, save, retrieve and delete plugin's defined data. Example: function `msvcs_session_get_ext_handle` (JSC00001013, line 519) is used to extract plugin dependent, session based state information.
	performing the processing of the identified component	Example: function `cpcd_xlate_packet` (JSC00001013, line 497), which is called while

	with the packet and the retrieved state information; and	processing packets of the session (JSC00001015, line 601), retrieves plugin dependent, session based state information (JSC00001013, line 519), and uses this information for packet processing (JSC00001013, line 532)
	storing state information relating to the processing of the component with the packet for use when processing the next packet of the message.	Example: function `cpcd_process_session_interest` (JSC00001004, line 185), which is called while processing the first packet of the session (JSC00001015, line 605), • allocates (JSC00001006, line 231), • updates (e.g., JSC00001006, line 247), and • saves (JSC00001007, line 288) plugin dependent, session based state information.
1. Preamble. A method in a computer system for processing a message having a sequence of packets,	Implicit incorporates by reference as though fully set forth herein the opening narrative and source code citations. **Evidence '163 C1 Pre(1)**	